ACEEE Int. J. on Information Technology , Vol. 3, No. 3, Sept 2013

Implementing True Zero Cycle Branching in Scalar and Superscalar Pipelined Processors

Mohit Gupta[1, a], Rishu Chaujar[1, b]
[1]Department of Engineering Physics, Delhi Technological University (Formerly Delhi College of Engineering),
Shahbad Daulatpur, Main Bawana Road, Delhi-110042, India.
Email: [a]mohit.gupta@dtu.co.in, [b]rishu.phy@dce.edu

***Abstract*— In this paper, we have proposed a novel architectural technique which can be used to boost performance of modern day processors. It is especially useful in certain code constructs like small loops and try-catch blocks. The technique is aimed at improving performance by reducing the number of instructions that need to enter the pipeline itself. We also demonstrate its working in a scalar pipelined soft-core processor developed by us. Lastly, we present how a superscalar microprocessor can take advantage of this technique and increase its performance.**

***Index Terms*—RISC, Computer Architecture, IPC, Branch Methodologies, Pipelining Processing, Loop Unrolling, Parallel processing, BTB, Branch folding, Cache Memory, costs, VHDL, FPGA, Xilinx ISE, Superscalar.**

I. Introduction

Pipelining improves computer performance by overlapping the execution of several different instructions. If no interactions exist between different parts of the pipeline, then several instructions can be in different stages of execution simultaneously. However, dependencies between instructions – particularly branch instructions – prevent the processor from realizing its maximum performance.

The power of the computer lies in its ability to dynamically alter its flow of control, choosing one of two (or more) execution paths. The basis of this decision-making ability is the conditional branch mechanism. As multiple studies have shown [1, 2, 3], branch-related instructions form a significant fraction of all instructions executed. For example, one out of every three instructions executed on the VAX and one out of every eight instructions executed on the IBM System/370 is a branch. Because of the high frequency of branches, the selection of particular branch architecture is a crucial decision in the design of a computer's instruction set. In a pipelined CPU, the impact of the branch design is magnified by the pipeline delays created by the branches [4, 5, 6]. Since branch instructions form a significant fraction of executed instructions and cause the maximum bottlenecks in any processor's performance, their design is therefore a crucial component of any architecture.

This novel technique of branching has been implemented in our design which is a 5-stage scalar pipelined soft-core processor, called MP. It essentially eliminates the need for a branch instruction to enter the pipeline (more than once). This means a "special" instruction will be executed only once and it will "tell" the processor that following 'x' instructions are to be executed until some 'y' condition is satisfied (or dissatisfied). Performance boosts of such a technique in loops can be easily imagined. The following sections describe this technique in detail including the motivation for its research, its implementation, advantages and limitations of using this technique, intuitive analysis of performance boosts and experimental results. Finally, our technique is compared with a few other similar schemes and how this can be implemented in a superscalar pipeline has been proposed.

A. Motivation

Consider for example, a "normal" loop, which has a branch instruction in the end as suppose 'loop again if sign flag is set'. This means that the body of the loop is to be executed again if the result of previous instruction is negative. Note the limitations involved here:

1) Result of only the previous instruction according to the logical sequence of the program can be checked in most current architectures. Since this instruction can potentially lead to branch hazards, it is important to improve pipeline performance at this stage. Hence, it will be beneficial if the result of any previous instruction according to the programmer's will could be checked. The MP allows this by giving the user the ability to select which instruction(s) can alter the conditional flags. Note that this is similar to the 'set if less than' instruction of the MIPS instruction set but in our case an extra instruction is generally not required to perform just a comparison.

2) Exit from the loop can only be granted at the last instruction. Exiting from in between would inevitably require another branch instruction leading to increased probability of branch hazards. It will be explained later how our technique is capable to branching at multiple sites in the logical sequence of the program, without requiring extra conditional branch instructions.

3) The loop instruction would need to enter the pipeline repeatedly in each iteration consuming significant resources like a slot in the fetch, decode and issue units, in reservation stations of the branch processing unit(s) and the reorder buffer. In some architectures, the results may even need to be broadcast from BPU onto the CDB. But it does the exact same thing every time, i.e. test the same conditional flags and decide whether to branch or not. Also it will receive the exact same value of the conditional flags every time except for the last iteration.

From above points it should be clear that loop instructions, or branch instructions in general, are very different from other 'useful' instructions and it might turn out to be

DOI: 01.IJIT.3.3.1275

more efficient if we implement them in a different way too. Furthermore, these are notorious in creating the most bottlenecks in processor performance. Our novel technique of 'true zero cycle branching' addresses the above points and removes these limitations while aiming for increased overall throughput.

II. Implementation

Implementing the true cycle branching technique shall primitively involve adding new instructions in the instruction set which explicitly exploit this facility. In later sections a method will be discussed by which the need to add new instructions in an ISA can also be eliminated. Consider a typical example of a program sequence as shown in Fig. 1, which utilizes 'true zero cycle branching'. This program finds all multiples of a number (in register r0) which are less than or equal to another number (in register r1) and calculates the number of terms (in register r4) of the sequence. Instructions have been written in a way to be easily understandable and mnemonics have been avoided here. It shows how these special instructions will be employed in a program sequence. Here the loop instruction means "keep iterating the following three instructions repeatedly until any of the zero or sign flags are set". The branch instruction enters the pipeline only once and the processor 'takes notes' from it and acts according to the logical sequence of the program.

```
Move   r0, 0x3  (r0 = 3)
Move   r1, 0xa3c8
Move   r2, r0
Move   r4, -1
Loop 3 when any of zero or sign flags are set
       *Subtract r3, r2, r1      (r3 = r1- r2)
       Add r2, r2, r0
       Add r4, r4, 1
Last:  jump to Last
```

Figure 1. Typical Program sequence.

As soon as the result of the subtract instruction is negative, the loop is terminated and the program resumes normal sequential execution. Note that the two add instructions following the subtract instruction will not be executed after the branch is taken in this example, but this is totally an implementation choice of the programmer, according to program specifications.

In simple words, instead of four only three instructions are now executed in each iteration. Although the performance of a program would depend mostly on the inner-most loop, this technique can also be extended to hierarchies of loops. Note that most loops that we program in high-level languages or even in assembly follow a stack-like pattern, i.e., the inner loops keep on stacking 'onto' the outer loop. Thus, for optimizing across hierarchies of loops, one can implement a small hardware stack dedicated for this purpose. Obviously, the specifications of the maximum stack depth etcetera will be decided upon by hardware resource availability factors and the information about the same will have to be provided to compiler and assembler developers, as exceeding the stack depth would overwrite the outer-most loop. It is then easily evident how this will work. The branch processing unit will produce the valid value (or values for superscalar processors) of the program counter (PC) in each cycle corresponding to the information only on the top on the stack, which will correspond to the inner-most loop. When the inner-most loop terminates, it is 'popped off' the stack and the information of the loop enclosing this loop will become 'active'.

The 'True Zero Cycle Branching' technique was implemented in the MP and it required very minimal extra resources. Fig. 2 lists possible varieties of this type of special instructions. The list can be further extended limited only by the complexities of more powerful instructions.

1) Loop following 'x' instructions when any of 'y' flags are set/ reset
2) Loop following 'x' instructions when all 'y' flags are set/ reset
3) Reiterate all following 'x' instructions until any of 'y' flags are set or reset
4) Reiterate all following 'x' instructions until all 'y' flags are set or reset
5) Loop following 'x' instructions 'y' times (requires an extra adder)
6) Branch to 'x' if any of 'y' flags are set/ reset (like try and catch blocks).
7) Branch to 'x' if all 'y' flags are set/reset.

Figure 2. Special Instruction Set

A. Advantages

This section will help the reader in truly realizing the performance boosts achievable with this technique.

1) It enables much more liberal use of powerful optimizing techniques and also makes them more effective. For example, consider the same program from Fig. 1 optimized by using the loop unrolling technique. This is shown in Fig. 3. The program has been written in a simplified way to demonstrate the effectiveness of the loop unrolling technique used along with 'true zero cycle branching'. Note that here the loop is unrolled once, i.e., two terms are calculated in each iteration and the subtract instructions are used as comparisons to alter the conditional flags. These two subtract instructions will affect the flags and the loop can be terminated immediately after any of these two instructions. This kind of optimization technique was not possible without the 'true zero cycle branching' technique. It should be noted that this led to optimization because one cycle per iteration would otherwise have to be wasted due to RAW data hazards and so it is an example of pipeline optimization. Many compilers are equipped with these kinds of optimization techniques.

2) It should also be easily incorporable in existing architectures as this will anyways require a new class of instructions. However, care will have to be taken to ensure that they do not conflict with current branch processing units.

DOI: 01.IJIT.3.3.1275

```
0.  Move r0, 0x5
1.  Move r1, 0xa3c8
2.  Move r2, r0
3.  Add r3, r0, r0
4.  Loop 4 when any of zero or sign flags
    are set
5.  *Subtract r4, r2, r1
6.  Add r2, r3, r0
7.  *Subtract r4, r3, r1
8.  Add r3, r2, r0
9.  Last:   jump to Last
```

Figure 3. Loop Unrolling

3) This class of branch instructions also does not require the use of speculative execution as the next instruction to be fetched is always known. The pipeline will need to be flushed only once, i.e. at the end of the loop as is always the case. Hence, this also acts as static branch prediction where the user/compiler would indicate whether the branch should be speculated as 'taken' or 'not taken'.

4) Implementing this technique for the MP required very minimal hardware resources like a few registers and an adder. It is easily seen that compared to speculative branching, our technique requires very minimal resources.

5) Potential performance improvements with this technique are discussed in the later section.

6) Although we included a new class of instructions to implement this technique, in some architectures it should also be possible to 'fold' branch instructions which have negative offset. This is because a branch backwards most likely indicates a loop and hence it should theoretically be possible to implement this, especially in simpler ISAs like MIPS. In MIPS, if the immediate field in a 'beq' instruction has a negative value, then it should be possible to extract all the required information from its first occurrence to resemble a 'true zero cycle branching' behavior. Typically the current PC, branch distance and branch condition are all that is required to 'fold' loop-type instructions and hence it is theoretically plausible to be able to accomplish this.

B. Limitations

1) A major setback to immediately implementing this technique is that since this is new, current-day compilers are not equipped to utilize its full potential. Since this would mostly rely on static scheduling, user involvement would be imperative to boost performance at least until smart compilers are developed which avail these facilities.

2) Even if compilers are equipped with utilizing these special instructions, users should be intimated with these technologies so that they try to make it easily detectable by the compiler. This way full potential of the performance boost can be achieved.

3) It was realized during its implementation that the Program Counter (PC) becomes a high fan-out and high fan-in signal. Since it is required that the updated values of PC (s) be available in each clock cycle, the fan-in cannot be reduced by pipelining. We, however, in certain cases may be able to reduce fan-out by register duplication, depending upon which all units require its values. This might depreciate timing performance and so some restrictions would have to be imposed on its use due to the need for optimizing its hardware implementation. In the MP, the only restriction we imposed is that the loop should have at least one instruction! This is obviously always going to be true.

C. Intuitive Analysis of Performance Boosts

Let us assume that every instruction consumes an equal number of clock cycles and hardware resources to complete, such that the absence of any one instruction would allow another instruction to execute in its place. Note that the above assumption is highly reasonable for scalar pipelined architectures like the MP.

Let 'm' be the number of instructions in the loop body excluding the branch instruction and 'n' be the number of times the loop iterates in actual execution. Since pipelined architectures would complete one instruction in each clock cycle, the total number of clock cycles to execute the loop for current architectures would be simply (m+1)*n (here '*' is used for multiplication), as the branch instruction would also enter the pipeline in each iteration. If 'true zero-cycle branching' were used, then a 'special branch instruction' (from Fig. 3) would be executed once and after that only the 'useful' instructions will be executed, i.e., m instructions will be executed in each loop iteration. Hence, the total number of clock cycles consumed would be m*n + 1. Therefore, the reduction in the number of instructions that need to be executed is

$$(m+1)*n - m*n - 1 = n - 1 \qquad (1)$$

This gives a rough indicative of the percentage speed-up(x) obtained by this technique as

$$x = (n-1)*100/[(m+1)*n]. \qquad (2)$$

For large number of iterations (n), the above equation saturates to a maximum value of 100/ (m+1). This shows that the maximum speed-up achievable in a loop depends largely on the number of instructions in the loop body, as expected. The graph in Fig. 4 shows the percentage speed-up as a function of the number of iterations of the loop (n) for typical values of the number of 'useful' instructions in the loop. It is easily seen from the graph that the percentage speed-up saturates at even small values of n ~ 25. The graph in Fig. 5 shows the maximum percentage speed-up obtainable as a function of the number of instructions in the loop (m).

It should be noted that the technique would also introduce other performance improvements which have not been taken into account in the above analysis, for e.g., it would eliminate or minimize the possibilities of outstanding conditional branches. Also it would prevent the Reorder Buffer (ROB) and reservation stations from filling up fast.

DOI: 01.IJIT.3.3.1275

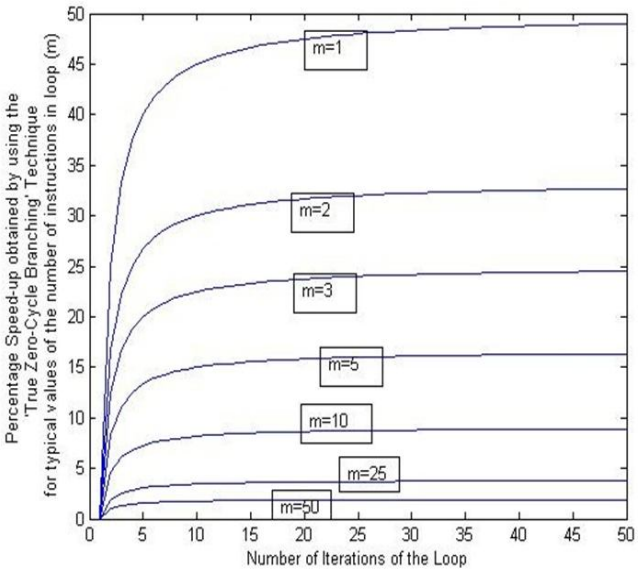

Figure 4. Graph of x =f(n) vs. n

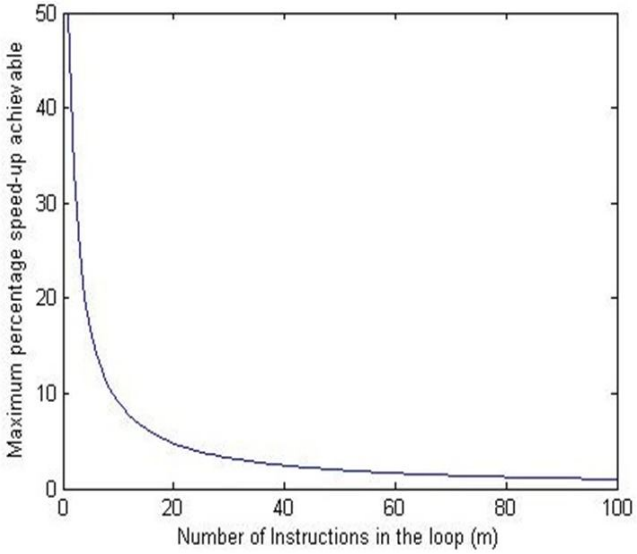

Figure 5. Maximum percentage speed-up achievable as a function of number of instructions in loop (m)

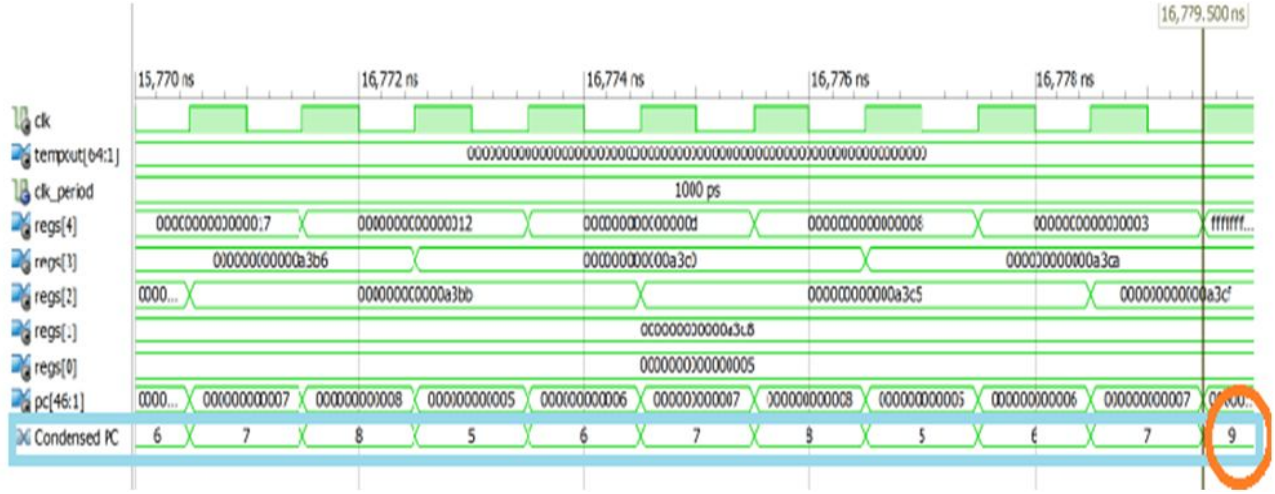

Figure 6. Simulation of MP executing the sample program. Note how the value of PC is changing.

III. Experimental Results

This section presents the actual simulation results of the MP design that implements this technique. It was designed to be implemented onto a Xilinx FPGA and the whole design was simulated with Xilinx ISim software. In this simulation the MP executes the program of Fig. 3. The loop has four instructions numbered 5 to 8. Fig. 6 shows the last ten clock cycles of the simulation. Note that the value of PC (Program Counter) remains between 5 and 8 until the loop is terminated after the last iteration. After the value of either register r2 or r3 has been tested to be greater, the loop immediately terminates and the value of PC becomes 9.

IV. Comparison to Other Schemes

The techniques used in the MP were aimed at creating a control-intensive design, rather than a memory-intensive one. This has led to a significant improvement in reducing cost of branches compared to traditional schemes in our design. The widely implemented and perhaps the most popular scheme, known as the *Branch Target Buffer (BTB) optimization* [5, 6, 7, 8, 9], maintains a selective Cache memory, associated with the instruction fetch portion of the pipeline, to hold branch target addresses and a few target branch instructions themselves, corresponding to each entry, in the improved scheme called *branch-folding*. When the pipeline decodes a branch instruction, it associatively searches the BTB for the address of that instruction. If it is found in the BTB, as it will be always for most loops after the first execution, the instruction is supplied directly to the pipeline and prefetching begins from the new path. This scheme achieves branching in zero-cycles for unconditional and some conditional branches, if condition codes are preset. However, if the condition codes are not already available (which will most likely be the case in loops), then the branch instruction will need to pass through the pipeline to complete, even though the target instruction may already have been fetched from the BTB. In this respect, our technique would outperform branch folding as in the MP the branch instructions are not required to enter the pipeline after first execution, thus saving slots in the issue unit and the branch processing unit (BPU) reservation station, which can then be used for other

DOI: 01.IJIT.3.3.1275

instructions. Moreover, fully-associative memories, as those required for the BTB, consume large chip areas for even small depths. 'True Zero Cycle Branching' on the other hand required minimal hardware resources in its implementation. It should be noted that the MP also has support for the traditional branch instruction types, and therefore, the programmer/compiler should not face any situation where 'True Zero cycle branching' technique for a loop is undesirable. Additionally, our scheme can also be implemented along with the BTB scheme so that the BTB can optimize global branches other than loops, with the only difference being that no entry will be made in the BTB for the special branch instructions relevant to our technique. In this way, more transistor-intensive associative memory logic can be effectively translated into simpler control logic, as the size of BTB can be marginally reduced and simultaneously loops can be better optimized.

Also, since most architectures limit the maximum number of outstanding branches in the BPU (generally 2-3), the processor can reach a performance bottleneck for very small loops, as the average issue rate for the loop will be reduced, which is not the case with the MP. The MP will be able to execute loops with even just a single instruction with zero cycle delays for branching. Additionally, in conventional architectures if the entry for a branch instruction is not found in the BTB, then branch prediction buffers are accessed to predict whether branch will be taken or not. In the MP, such prediction schemes are not required since it is most likely that the loop body will be executed at least once.

The AT&T CRISP processor [9, 10] uses a technique called *branch folding* that can reduce the branch penalty to zero. In this scheme, the pipeline is broken into a Prefetch and Decode Unit (PDU) and an execution unit (EU). The fetch-and-decode unit reads encoded instructions from memory, which may be as short as 16-bits and places the canonical fixed 192-bit length decoded instructions into a special instruction cache, thus effectively decoupling the PDU from the EU. The PDU recognizes when a branch instruction follows a non-branch instruction and "folds" the branch instruction into the non-branch instruction, thereby eliminating the branch instruction from execution pipeline. When a folded conditional instruction executes, the execution unit selects one of the two next-address paths and the address of the other path proceeds down the pipeline along with the executing instruction. Note that although the MP also stores addresses of both execution paths, it does so only in the BPU. Whereas in the CRISP, the next-address proceeds down the entire execution pipeline of the processor. Additionally, branch folding also requires a large instruction cache due to large size of the decoded instructions. This scheme also involves significantly more complicated hardware [9].

V. Proposed implementation in a superscalar microprocessor model

Indubitably, implementing this technique in a superscalar [11] processor would be many times complex than in a scalar pipelined processor. This section presents our analysis of the problem and how it can be dealt with. It should be noted that although it is a daunting task, the performance boosts expected are also magnified if it is successfully implemented. This work is currently in progress and hence, we will present a few challenges that we encountered.

A. Issues

1) Just like other complexities involved while developing superscalar pipelines, increasing issue rate of the architecture leads to exponentially more complex implementation of this technique. The more instructions that the pipeline is capable of issuing in each cycle, the more instructions (or instruction pointers) would the BPU need to produce in each cycle. The degree of complexity of the final design would also depend on a few design choices. For example, whether a designer choses to make the design capable of unrolling a loop more than once in a single clock cycle is very important. This will definitely have a huge impact on the IPC (instructions per clock cycle), especially if a program consists of loops with very few instructions (which is how it is strongly dependent on the issue rate). Also, if it is capable of unrolling a loop more than once, then it should be able to compare more than one result against the loop exit condition. In most technologies, we expect that including this ability would have an adverse effect on the timing performance of the device. For these reasons, we have decided to not include this capability in our design which is targeted for FPGA devices.

2) Since the device is now capable of executing more than one instruction per clock cycle, if more than one instruction capable of updating flags enter in each clock cycle then the IPC would drop down to nearly one (equivalent to a scalar pipeline) in case the designer did not include the capability to compare more than one results per clock cycle against loop exit conditions. And including this ability, as discussed above, can exacerbate timing performance in addition to requiring more hardware resources.

3) To implement the 'true zero cycle branching' technique in a superscalar pipeline, the retire stage will have to be designed differently too. This is because we do not want to make an entry for every time the loop iterates so as to increase the effective retire rate. This can however be dealt rather easily by appending a 'speculative depth' field to all instructions that are part of a loop. Then the BPU can send signals to the reorder buffer to update a 'correctly speculated depth' field and then the latter can compare against the 'speculative depth' fields of the instructions in its retire window to decide which instructions can retire on the next clock edge.

4) As discussed before for a scalar pipeline, the Program Counter (PC) signal may have a very high fan-out and fan-in. Moreover, a superscalar design will probably have as many different PC signals as the issue rate of the pipeline.

5) Due to register renaming, keeping track of the results is a daunting task. This is because the loop instruction enters the pipeline only once and hence extra measures need to be taken to obtain updated values of the various renamed

DOI: 01.IJIT.3.3.1275

registers that the branch needs to access. This is explained in the next section.

B. Implementation

This section discusses our proposed implementation which should make this technique possible in a superscalar architecture. In this implementation, even a special instruction set is not required to achieve 'true zero cycle branching' and the 'branch folding' approach discussed before will be used.

Branch instructions with a negative offset can be removed from the pipeline and information regarding their branch target, current PC and the loop exit condition can be sent directly to a 'special' Branch Processing Unit (BPU) during its first occurrence. This has another dimension of complexity involved with it if a loop has very few instructions in it. The designer will have to take corrective measures in case the same branch instruction is able to enter the pipeline again due to very small loops and/or high issue rate.

After the BPU has received the required information, it will need to listen to broadcasts on the Common Data Buses (CDBs) and check if a loop exit condition is satisfied. But to do this, it should have the tags (or physical register addresses) of the corresponding registers which will be modified constantly during the decode stages for new writes. A simple but a little expensive fix is to also send logical register addresses to the BPU and build two more read ports to the Remap File so that BPU can listen to them and accordingly update the tags of the concerned registers.

Now that the BPU has all the information to keep unrolling or to cycle through the loop, all it has to do is to send signals to the reorder buffer to indicate whenever an instruction(s) is determined correctly speculated or to exit the loop and flush the pipeline if determined incorrectly speculated. Hence, this unit will only listen to broadcasts on the CDB but not write to it as it cannot produce any results to be written into the reorder buffer which would otherwise reduce the retire rate. The reorder buffer will need to update its 'correctly speculated depth' fields and accordingly decide whether instructions in the retire window are to be retired or not.

VI. Conclusions

We have presented a whole new way of dealing with conditional branches whose benefits are easily seen. It targets the most basic thing to improve performance, i.e., reduce the number of instructions that need to enter the pipeline itself while simultaneously ensuring that there are no branch latencies. Some basic limitations of the conventional methods to deal with conditional branches were pointed out and how this novel technique addresses them has been discussed. A mathematical analysis provided indicative figures of how much performance improvement can be expected with this technique. It was demonstrated how this method is in its true sense – 'true zero cycle branching' with an example of the MP. This way we proved that this technique can in fact be readily employed in existing architectures, as was the case with MP. A different approach which does away with the requirement of having a special instruction set to implement this technique was discussed which reinforces our assertion that this technique should be easily incorporable in existing architectures as well. We also compared our scheme with a few other well-known techniques and demonstrated how it could out-perform them. The choice of which techniques to use, however, depends on the performance requirements and cost constraints of a particular processor design.

Finally, the future scope of our work which involves implementation of 'true zero cycle branching' in a superscalar pipeline was discussed. Some issues were pointed out and an implementation is proposed. Even though it looks expensive to implement this technique in a superscalar pipeline in terms of hardware resources required, the expected performance boosts also cannot be overlooked in this regard.

References

[1] Joel S. Emer and Douglas W. Clark, "A characterization of processor performance in the VAX-11/780", in Proceedings, The llth Annual Symposium on Computer Architecture, pages 301-310, ACM and the IEEE Computer Society, June 1984.

[2] Leonard Jay Shustek, "Analysis and Performance of Computer Instruction Sets", PhD thesis, Stanford University, January 1978.

[3] Knuth, D. E. (1971), "An empirical study of FORTRAN programs". Softw: Pract. Exper., 1: 105–133. doi: 10.1002/spe.4380010203.

[4] Peter Kogge, "The Architecture of Pipelined Computers", McGraw-Hill Book Company, 1981.

[5] J. Hennessy and D. Patterson, "Computer Architecture, A Quantitative Approach", Morgan Kaufmann Publishers, Inc., 1990.

[6] Patterson, D. A., Hennessy, J. L., "Computer Organization and Design: The Hardware/Software Interface", 2nd edition, Morgan Kaufmann Publishers, San Francisco, CA, 1998.

[7] Lee, J.K.F.; Smith, A.J., "Branch Prediction Strategies and Branch Target Buffer Design," Computer, vol.17, no.1, pp.6-22, Jan. 1984. doi: 10.1109/MC.1984.1658927.

[8] Perleberg, C.H.; Smith, A.J., "Branch target buffer design and optimization," Computers, IEEE Transactions on, vol.42, no.4, pp.396-412, Apr 1993. doi: 10.1109/12.214687

[9] Lalja, D. J.; "Reducing the branch penalty in pipelined processors," Computer , vol.21, no.7, pp.47-55, July 1988. doi: 10.1109/2.68

[10] D. R. Ditzel and H. R. McLellan "Branch Folding in the CRISP Microprocessor: Reducing Branch Delay to Zero", Proceedings of the 14th International Symposium on Computer Architecture, pp.2 -9 1987.

[11] J. Shen and M. Lipasti, "Modern Processor Design – Fundamentals of Superscalar Processors", Tata McGraw-Hill Company Limited, 2005.

DOI: 01.IJIT.3.3. 1275